

# OperatingSystem 3조 보고서

- 권정태, 김도형, 윤재선, 허성현

---

## 목차

### 1. 서론

- 프로젝트 선정이유
- 프로젝트 개요

### 2. 설계 개요

- xv6 동작 방식 설명
- slab 구조체에 대한 설명
- 프로젝트 메모리 할당방식 설명

### 3. 상세 구현 및 설명

- Bitmap
- slabinit
- slab memory allocation
- slab memory deallocation
- debug 관련 함수

### 4. Test 및 구현

- linux 환경에서 test하는 방법
- docker 환경에서 test하는 방법
- Test 프로그램 설명
- test 실행 결과

### 5. Trouble shooting

### 6. 프로젝트 일지

### 7. 결론

## ▼ 1. 서론

### ▼ 프로젝트 선정이유

수업에서는 운영체제가 프로그램 실행 시 필요한 메모리 공간을 제공하고 관리하는 기능을 배운 바 있습니다. 메모리 할당은 주로 정적 할당과 동적 할당으로 나뉘며, 동적 할당은 프로그램 실행 중 메모리를 요청하고 해제하는 과정을 포함합니다. 이 과정에서 내부 단편화와 외부 단편화 문제가 발생할 수 있습니다. 특히, xv6는 기본적으로 4KB의 고정된 크기로 메모리를 할당하기 때문에 4KB보다 작은 메모리 할당이 많이 발생할 때 심각한 메모리 낭비로 이어질 수 있습니다. 저희 팀은 이러한 비효율적인 페이징 방식과 내부 단편화 문제를 해결하기 위해 OS에서 메모리 할당 및 해제 부분을 개선하고자 프로젝트를 선정하고 구현했습니다.

### ▼ 프로젝트 개요

프로젝트의 목표는 xv6 운영체제에 다양한 바이트 단위의 메모리 할당기를 구현하는 것입니다. 이를 위해 각 고정 크기의 바이트(8, 16, 32, 64, 128, 256, 512, 1024, 2048 바이트)에 대해 개별 할당기를 구현합니다. 이 프로젝트에서는 4KB 크기의 페이지를 비트맵으로 사용하여 할당된 슬랩을 구별합니다. 비트맵은 각 슬랩 내에서 할당된 객체를 추적하고 관리하는 데 사용되며, 할당된 객체의 상태를 효율적으로 관리하고 빠르게 검색할 수 있습니다. 이를 통해 메모리 할당 과정에서 발생하는 내부 단편화 문제를 해결하고, 메모리 사용 효율을 높이는 것을 목표로 합니다.

## ▼ 2.설계 개요

### ▼ Xv6 동작 방식 설명

xv6은 x86 arch기반 멀티프로세서 시스템에서 동작하며, 부팅 과정에서 BIOS가 실행되어 hardware를 초기화하고, boot loader 인 bootasm.S 가 로드됩니다.

bootasm.S 는 프로세서 초기화, 메모리 관리, interrupt 처리 등의 기능을 수행하며, 이후 main.c 의 main() function 이 실행되어 운영체제의 핵심 기능을 수행합니다.

xv6는 프로세스 생성, 스케줄링, 메모리 관리 등의 기본적인 기능을 제공합니다. 내부적으로 프로세스 creation에는 fork() system call 을 사용하며, exec() system call 을 통해 새로운 프로그램을 실행할 수 있습니다.

스케줄링은 round-robin 방식을 사용하며, 우선 순위가 높은 프로세스를 먼저 실행합니다. paging 및 paging table 을 사용하여 물리 메모리를 효율적으로 사용합니다.

xv6는 간단한 파일 시스템을 구현하고 있습니다.파일 시스템은 inode 기반으로 구현되며, 파일 생성, 삭제, 읽기, 쓰기 등의 기능을 제공합니다.디스크 입출력은 IDE driver를 통해 수행되며, 버퍼 캐싱 기법을 사용하여 성능을 향상시킵니다.

xv6는 키보드, 화면, 디스크 등의 입출력 장치를 지원합니다.입출력 처리는 인터럽트 기반으로 수행되며, 인터럽트 핸들러가 입출력 요청을 처리합니다. 입출력 처리 과정에서 driver module 이 사용되며, 이를 통해 하드웨어와 운영체제 간 통신이 이루어집니다.

xv6는 32bit virtual address space 을 사용하여 최대 4GB의 memory 를 다룰 수 있습니다. memory 는 커널 전용 영역과 사용자 프로세스 전용 영역으로 나뉩니다.커널 코드는 연속된 physical memory space 에 load 됩니다.

xv6는 기본적으로 paging 기법을 사용하여 memory를 관리합니다.프로세스에 메모리를 할당하는 주요 시스템콜은 sbrk()이며, sbrk() 시스템콜은 process 의 heap 영역을 확장하여 메모리를 할당합니다.

xv6의 main function 의 호출 과정은 다음과 같습니다.

#### 1. boot strap 과정

- BIOS 에 의해 부팅되며, boot loader 인 bootasm.S 와 bootmain.c 가 실행된다.
- bootmain.c 에서 kernel image 를 memory 에 load하고, main() function 을 호출합니다.

#### 2. main() function 호출

- main() function은 xv6 kernel의 진입점이며, physical memory 관리, process / file system / device driver 초기화/ init process 생성 및 execution 등의 작업을 수행합니다.

#### 3. init process 실행

- init process 는 /init 파일을 실행하며, 시스템 초기화 작업이 포함되어 있습니다.
- xv6 kernel 은 user process 를 실행하는 스케줄러 역할을 실행합니다.

#### 4. kernel 과 user process 간 switching

- system call , exception , interrupt 발생 시 user process 에서 kernel 로 제어가 전환됩니다.

- b. `kernel` 은 모든 `interrupt` 처리하며, 이는 대부분의 경우 `kernel` 만이 필요한 권한과 상태를 가지고 있기 때문입니다.

### ▼ slab 구조체에 대한 설명

```
#define NSLAB 9 // {8, 16, 32, 64, 128, 256, 512, 1024, 2048}
#define MAX_PAGES_PER_SLAB 100

struct slab {
    int size;
    int num_pages;
    int num_free_objects;
    int num_used_objects;
    int num_objects_per_page;
    char *bitmap;
    char *page[MAX_PAGES_PER_SLAB];
};
```

#### 구조체 멤버 설명:

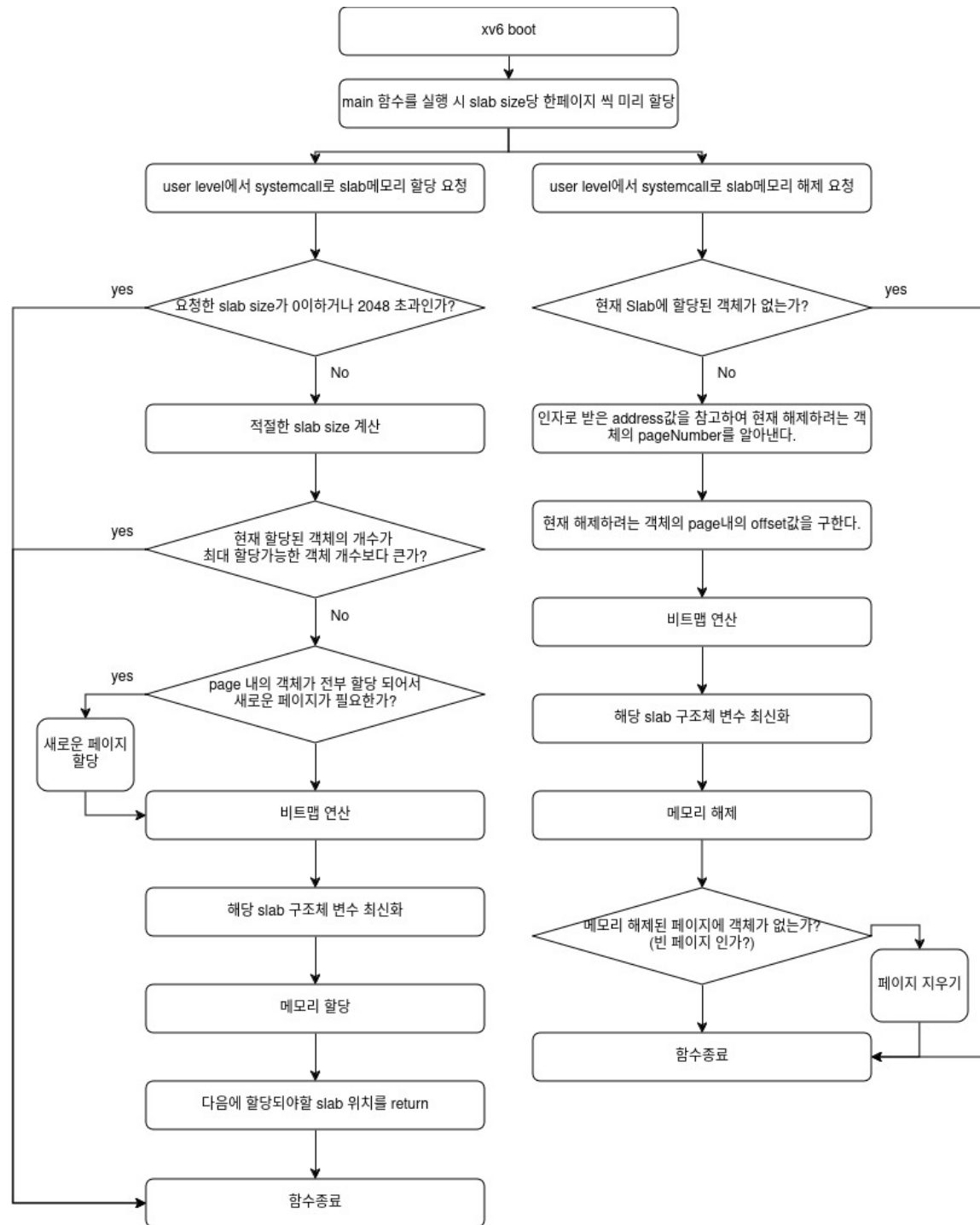
- `size` : 이 slab에서 각 객체의 크기.
- `num_pages` : 현재 이 slab에 할당된 페이지 수.
- `num_free_objects` : 이 slab 내에서 사용 가능한 객체 수.
- `num_used_objects` : 현재 사용 중인 객체 수.
- `num_objects_per_page` : 한 페이지에 들어갈 수 있는 객체 수.
- `bitmap` : slab 페이지 내의 객체 할당 상태를 추적하는 비트맵.
- `page` : 이 slab에 할당된 페이지에 대한 포인터 배열 ( `MAX_PAGES_PER_SLAB` 만큼).

\*MAX\_PAGES\_PER\_SLAB 값을 100으로 설정해서 slab별로 할당할 수 있는 페이지 수는 100개가 최대입니다.

\*NSLAB은 9로 설정을 하였는데 이는 총 SLAB의 개수가 9임을 의미하고, 각 SLAB은 8, 16, 32, 64, 128, 256, 512, 1024, 2048byte로 구성되어 있습니다.

### ▼ 프로젝트 메모리 할당방식 설명

- User level에서 메모리 할당과 해제에 해당되는 block-diagram



### • 초기 설정 단계 블록 다이어그램 설명

Slab allocate를 하기전에 page가 하나씩 할당이 되어 있어야 합니다. 따라서 xv6가 부팅이 되고 main함수를 호출을 할 때 앞선 과정이 필요합니다. 이 과정은 slabinitt()함수로 구현을 하였고 slabinitt()함수를 main 함수 실행시 호출될 수 있도록 만들었습니다.

slabinitt()함수는 각 slab 별로 한 페이지 씩 할당을 미리 해주는 함수입니다.

다음은 main함수의 일부분 입니다.

```

#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"

static void startothers(void);
static void mpmain(void) __attribute__((noreturn));
extern pde_t *kpgdir;
extern char end[]; // first address after kernel loaded from ELF file

// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    lapicinit(); // interrupt controller

```

```

    seginit();          // segment descriptors
    picinit();          // disable pic
    ioapicinit();       // another interrupt controller
    consoleinit();      // console hardware
    uartinit();         // serial port
    pinit();            // process table
    tvinit();           // trap vectors
    binit();            // buffer cache
    fileinit();         // file table
    ideinit();          // disk
    startothers();      // start other processors
    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
    slabinit();
    userinit();         // first user process
    mpmain();           // finish this processor's setup
}

```

밑에서 세번째 줄을 보면 slabinit()함수가 선언되어 있는걸 확인 할 수 있습니다.

- **slab 메모리 할당 블록 다이어그램 설명**

slab메모리 할당은 userlevel에서 이루어지고, 메모리 할당 함수는 kmalloc으로 정했습니다.

해당 함수는 실제로 메모리를 접근을 해야하는 함수이기 때문에 syscall로 구현을 했습니다. 전체적인 흐름은 다음과 같습니다. kmalloc을 호출할 때 파라미터로 유저가 할당하고 객체의 사이즈를 전달을 합니다. kmalloc함수에서는 객체의 크기가 0보다 같거나 작는지 2048보다 큰 지 판단을 합니다. 만약에 앞의 조건이 참이라면 slab allocator로 메모리를 할당 할 수 없기 때문에 함수를 종료시킵니다. slab의 크기가 8,16,32 .. 2048등으로 고정되어있는데 만약에 user-level에서 17 byte 를 할당을 한다면 slab allocator는 적어도 32byte 크기의 메모리를 할당을 해야하므로 slabsize에 따라 어떤 slab을 사용할건지 정하는 logic을 구현했습니다. 할당하려는 page에 객체가 꼭 차있는 경우에는 새롭게 페이징을 할 수 있도록 구현을 하였습니다. 새롭게 page를 할때는 kalloc 함수를 사용합니다.

다음은 kalloc함수의 code입니다.

```

struct run {
    struct run *next;
};

struct {
    struct spinlock lock;
    int use_lock;
    struct run *freelist;
} kmem;

char*
kalloc(void)
{
    struct run *r;

    if(kmem.use_lock)
        acquire(&kmem.lock);
    r = kmem.freelist;
    if(r)
        kmem.freelist = r->next; // first element of linked list changed
    if(kmem.use_lock)
        release(&kmem.lock);
    return (char*)r;
}

```

Physical memory의 Free frame에 접근을 할 때는 lock을 한 후에 linked list의 제일 앞에 있는 freeframe의 위치를 r에 저장하고 linked list의 제일 앞 위치를 수정을 합니다.

최종적으로 kalloc에서는 freeframe의 위치를 반환을 합니다.

memory allocation이 잘 되었는지 확인하기 위해서 bitmap을 0으로 1로 바꾸는 작업을 하고, 객체가 한 개 할당이 되었으므로 해당 slab 구조체의 변수를 최신화를 합니다. 다음에 할당되어야할 slab위치를 return하며 함수를 종료합니다.

- **slab 메모리 해제 블록 다이어그램 설명**

slab메모리 해제는 다음과 같은 과정으로 이루어집니다. slab 메모리 해제는 kfree함수로 구현을 했습니다. 이 함수는 해제하려는 slabsize와 해제하려는 메모리의 위치를 인자로 받아와서 메모리 해제를 진행을 합니다.

먼저 slab에 객체가 한개도 없으면 함수를 종료합니다.

slab에 객체가 존재한다면 인자로 받은 address값을 참고하여 현재 해제하려는 객체의 pageNumber를 알아냅니다. 다음으로 page내의 객체의 offset을 구합니다. offset값을 토대로 메모리가 비트맵의 어느 위치에 있는지 알아내고 해당 비트맵의 비트를 1에서 0으로 바꿉니다. 다음으로 해당 slab 구조체 변수를 최신화를 하고 메모리를 해제를 합니다. 만약에 메모리 해제를 한 뒤에 해제한 page에 객체가 있는지 검사를 하고 page내에 아무 객체가 없다면 page를 kfree함수로 지웁니다.

kfree함수는 다음과 같습니다.

```
void
kfree(char *v)
{
    struct run *r;

    if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    {
        cprintf("%d %d %d\n", (uint)v%PGSIZE, v<end, V2P(v)>=PHYSTOP);
        panic("kfree");
    }

    // Fill with junk to catch dangling refs.
    memset(v, 1, PGSIZE); // garbage value

    if(kmem.use_lock)
        acquire(&kmem.lock);
    r = (struct run*)v;
    r->next = kmem.freelist;
    kmem.freelist = r;
    if(kmem.use_lock)
        release(&kmem.lock);
}
```

kfree함수는 해제하려는 page의 위치를 인자로 받아옵니다.

page의 위치가 올바르지 않다면 panic 함수를 호출해 에러메세지를 출력하고 종료합니다.

page 위치가 올바르다면 메모리를 해제하고 해제된 page를 freelist의 가장 앞부분에 넣어줍니다.

### ▼ 3. 상세 구현 및 설명

#### ▼ Bitmap



**set\_bit(int num, int i)**

```
int set_bit(int num, int i)
{
    return num | (1 << i);
}
```

이 함수는 주어진 정수 `num`의 `i` 번째 비트를 1로 설정하여 반환합니다. 이를 위해 `1 << i`를 사용하여 `i` 번째 비트만 1이고 나머지는 0인 값을 만들고, 이를 `num`과 OR 연산하여 `i` 번째 비트를 1로 설정합니다.



### get\_bit(int num, int i)

```
int get_bit(int num, int i)
{
    return ((num & (1 << i)) != 0);
}
```

이 함수는 주어진 정수 `num`의 `i` 번째 비트의 값을 반환합니다. `num & (1 << i)`를 통해 `i` 번째 비트의 값을 추출하고, 이 값이 0이 아니면 true(1)를 반환합니다.



### clear\_bit(int num, int i)

```
int clear_bit(int num, int i)
{
    int mask = ~(1 << i);
    return num & mask;
}
```

이 함수는 주어진 정수 `num`의 `i` 번째 비트를 0으로 설정하여 반환합니다. `~(1 << i)`를 통해 `i` 번째 비트만 0이고 나머지는 1인 마스크를 만들고, 이를 `num`과 AND 연산하여 `i` 번째 비트를 0으로 설정합니다.



### getnumofBits(unsigned int n)

```
unsigned int getnumofBits(unsigned int n)
{
    unsigned count = 0;
    while (n != 0)
    {
        n >>= 1;
        count += 1;
    }
    return count;
}
```

이 함수는 주어진 정수 `n`의 비트 수를 반환합니다. `n`을 오른쪽으로 1비트씩 shift하면서 `count`를 1씩 증가시킵니다. 이 과정을 `n`이 0이 될 때까지 반복하면 `n`의 비트 수를 구할 수 있습니다.



### nextPowerOf2(unsigned int n)

```
unsigned int nextPowerOf2(unsigned int n)
{
    if (n >= 0 && n < 8)
        return 8;
    if (n && !(n & (n - 1)))
        return n;
    return 1 << getnumofBits(n);
}
```

이 함수는 주어진 정수 `n` 보다 크거나 같은 2의 거듭제곱 수를 반환합니다. 먼저 `n` 이 0부터 7 사이인 경우 8을 반환합니다. 그 외의 경우, `n` 이 2의 거듭제곱 수인지 확인하고, 그렇다면 `n` 을 반환합니다. 그렇지 않은 경우, `getnumofBits(n)` 을 통해 구한 비트 수에서 4를 빼고 2를 거듭제곱한 값을 반환합니다.



### getslabIdx(unsigned int n)

```
unsigned int getslabIdx(unsigned int n)
{
    unsigned slabIdx=getnumofBits(n)-4;
    return slabIdx;
}
```

이 함수는 주어진 정수 `n` 의 slab 인덱스를 반환합니다. `getnumofBits(n)` 을 통해 구한 비트 수에서 4를 빼면 slab 인덱스를 얻을 수 있습니다.



### returnOffset(int row, int column)

```
int returnOffset(int row, int column)
{
    return 8 * row + column;
}
```

`returnOffset(int row, int column)` 함수: 이 함수는 주어진 행(`row`)과 열(`column`)을 이용하여 offset 값을 계산하여 반환합니다. offset은

`8 * row + column` 으로 계산됩니다.





### setBitmap(int slabIdx)

```
int setBitmap(int slabIdx)
{
    struct slab *s;
    s = &stable.slab[slabIdx];
    for (int j = 0; j < PGSIZE; j++)
    {
        if (s->bitmap[j] == 0xFF)
            continue;
        for (int k = 0; k <= 7; k++)
        {
            if (!(s->bitmap[j] & (1 << k)))
            {
                s->bitmap[j] = set_bit(s->bitmap[j], k);
                return returnOffset(j, k);
            }
        }
    }
    return 0; // Unable to find empty space of bitmap
}
```

이 함수는 주어진 slab 인덱스( `slabIdx` )에서 사용 가능한 offset을 찾아 반환합니다. `stable.slab[slabIdx]` 에 해당하는 slab 구조체를 가져와, 각 비트맵 엔트리를 검사하여 비어있는 공간을 찾습니다. 비어있는 공간을 찾으면 `set_bit` 함수를 사용하여 해당 비트를 1로 설정하고, 계산된 offset을 반환합니다. 사용 가능한 공간을 찾지 못한 경우 0을 반환합니다.



### getRow(int offset)

```
int getRow(int offset)
{
    return offset / 8;
}
```

이 함수는 주어진 offset 값에서 행 번호를 계산하여 반환합니다. offset을 8로 나눈 몫이 행 번호입니다.



### getColumn(int offset)

```
int getColumn(int offset)
{
    return offset % 8;
}
```

이 함수는 주어진 offset 값에서 열 번호를 계산하여 반환합니다. offset을 8로 나눈 나머지가 열 번호입니다.



### clearBitmap(int slabIdx, int offset)

```
bool clearBitmap(int slabIdx, int offset)
{
    struct slab *s;
    s = &stable.slab[slabIdx];
    bool checkbit = true;
    int row = getRow(offset);
    int column = getColumn(offset);
    if (get_bit(s->bitmap[row], column))
        s->bitmap[row] = clear_bit(s->bitmap[row], column);
    else
        checkbit = false;
    return checkbit;
}
```

이 함수는 주어진 slab 인덱스( `slabIdx` )와 offset 값에 해당하는 비트를 0으로 설정합니다. `getRow` 와 `getColumn` 함수를 사용하여 행과 열 번호를 계산하고, `clear_bit` 함수를 사용하여 해당 비트를 0으로 설정합니다.비트가 정상적으로 0으로 설정되었다면 `true` 를, 그렇지 않다면 `false` 를 반환합니다.



### checkEmpty(int startOffset, int endOffset, int slabIdx)

```
bool checkEmpty(int startOffset, int endOffset, int slabIdx)
{
    struct slab *s;
    s = &stable.slab[slabIdx];
    bool empty = true;
    int startRow = getRow(startOffset);
    int startColumn = getColumn(startOffset);
    int endRow = getRow(endOffset);
    int endColumn = getColumn(endOffset);
    // cprintf("startOffset: %d, endOffset %d\n",startOffset,endOffset);
    // cprintf("startRow:%d endRow:%d\n",startRow,endRow);
    for (int i = startRow; i <= endRow; i++)
    {
        for (int j = startColumn; j <= endColumn; j++)
        {
            if (get_bit(s->bitmap[i], j))
            {
                empty = false;
                break;
            }
        }
        if (empty == false)
            break;
    }
    return empty;
}
```

이 함수는 주어진 slab 인덱스( `slabIdx` )에서 `startOffset` 부터 `endOffset` 까지의 범위가 모두 비어있는지 확인합니다. `getRow` 와 `getColumn` 함수를 사용하여 시작 offset과 끝 offset에 해당하는 행과 열 번호를 계산합니다.계산된 범위 내의 모든 비트가 0인지 확인하고, 모두 0이면 `true` 를, 그렇지 않으면 `false` 를 반환합니다.



### checkNetpage(int slabIdx)

```
int checkNewpage(int slabIdx)
{
    struct slab *s;
    s = &stable.slab[slabIdx];
    int cnt = 0;
    bool find0 = false;
    for (int i = 0; i < PGSIZE; i++)
    {
        for (int j = 0; j <= 7; j++)
        {
            if (get_bit(s->bitmap[i], j))
                cnt++;
            else
            {
                find0 = true;
                break;
            }
        }
        if (find0)
            break;
    }
    if (cnt % s->num_objects_per_page == 0)
    {
        int startOffset = (cnt / s->num_objects_per_page) * s->num_objects_per_page;
        int endOffset = (startOffset + s->num_objects_per_page) - 1;
        if (checkEmpty(startOffset, endOffset, slabIdx))
        {
            return startOffset;
        }
    }
    return 0;
}
```

이 함수는 주어진 slab 인덱스( `slabIdx` )에서 새로운 페이지를 찾아 반환합니다. 먼저 slab 구조체에서 사용 중인 오프셋 수를 계산합니다. 사용 중인 오프셋 수가 slab의 `num_objects_per_page` 의 배수이고, 해당 범위가 모두 비어있다면 시작 오프셋을 반환합니다. 새로운 페이지를 찾지 못한 경우 0을 반환합니다.



getpageNum(int slabIdx)

```
int getpageNum(int slabIdx)
{
    struct slab *s;
    s = &stable.slab[slabIdx];
    int page = 0;
    // size 2048 - 1024
    if (slabIdx >= 7)
    {
        for (int i = 0; i < PGSIZE; i++)
        {
            for (int j = 0; j <= 7; j += (PGSIZE / s->size))
            {
                for (int k = j; k < j + (PGSIZE / s->size); k++)
                {
                    if (get_bit(s->bitmap[i], k))
                    {
                        page++;
                        break;
                    }
                }
            }
        }
    }
    // size 8 - 512
    else
    {
        for (int i = 0; i < PGSIZE; i += (512 / s->size))
        {
            for (int j = i; j < i + (512 / s->size); j++)
            {
                if (s->bitmap[j] != 0x00)
                {
                    page++;
                    break;
                }
            }
        }
    }
    return page;
}
```

이 함수는 slab 크기에 따라 다른 방식으로 페이지 수를 계산합니다. 크기가 큰 slab의 경우 각 비트맵 엔트리를 세부적으로 검사하지만, 크기가 작은 slab의 경우 slab 크기 단위로 건너뛰면서 검사합니다.

## ▼ Slabinit

### Stable 구조체 구현

```
struct {
    struct spinlock lock; //stable에 한번에 한개의 current flow만 접근 가능하게 설정
    struct slab slab[NSLAB];
} stable;
```

여러 slab(8, 16, 32, 64, 128, 256, 512, 1024, 2048)을 할당하기 위한 구조체인 stable구현  
slabinit()함수 code

### slabinit 함수의 구현

```

void slabinit()
{
    acquire(&stable.lock);
    stable.slabs[0].size=8;
    stable.slabs[0].num_objects_per_page=PGSIZE/stable.slabs[0].size;
    stable.slabs[0].num_used_objects=0;
    stable.slabs[0].num_free_objects=stable.slabs[0].num_objects_per_page*64;
    //allocate one page for bitmap, allocate one page for slab cache
    stable.slabs[0].bitmap=stable.slabs[0].page[0];
    stable.slabs[0].bitmap=kalloc();
    memset(stable.slabs[0].bitmap,0,PGSIZE);
    stable.slabs[0].page[1]=kalloc();
    stable.slabs[0].num_pages=1;
    release(&stable.lock);

    acquire(&stable.lock);
    for(int i=1;i<NSLAB;i++)
    {
        stable.slabs[i].size=stable.slabs[i-1].size*2;
        stable.slabs[i].num_objects_per_page=PGSIZE/stable.slabs[i].size;
        stable.slabs[i].num_used_objects=0;
        stable.slabs[i].num_free_objects=stable.slabs[i].num_objects_per_page*MAX_PAGES_PER_SLAB;
        //allocate one page for bitmap, allocate one page for slab cache
        stable.slabs[i].bitmap=stable.slabs[i].page[0];
        stable.slabs[i].bitmap=kalloc();
        memset(stable.slabs[i].bitmap,0,PGSIZE);
        stable.slabs[i].page[1]=kalloc();
        stable.slabs[i].num_pages=1;
    }
    release(&stable.lock);
}

```

## slabinit() 함수에 대한 설명

- 락 획득: `acquire(&stable.lock)` 를 통해 락을 다시 획득합니다.
- 반복문을 통해 각 slab 초기화:
  - slab 크기 설정: `stable.slabs[i].size = stable.slabs[i - 1].size * 2;` 로 이전 slab 크기의 두 배로 설정합니다.
  - 페이지 당 객체 수 계산: `stable.slabs[i].num_objects_per_page = PGSIZE / stable.slabs[i].size;` 로 페이지 당 저장할 수 있는 객체 수를 계산합니다.
  - 사용된 객체 수 초기화: `stable.slabs[i].num_used_objects = 0;` 로 사용된 객체 수를 0으로 초기화합니다.
  - 여유 객체 수 설정: `stable.slabs[i].num_free_objects = stable.slabs[i].num_objects_per_page * MAX_PAGES_PER_SLAB;` 로 초기 여유 객체 수를 설정합니다.
  - 비트맵 초기화: `stable.slabs[i].bitmap = kalloc();` 로 비트맵 메모리를 할당하고 `memset` 을 통해 초기화합니다.
  - 첫 페이지 할당: `stable.slabs[i].page[1] = kalloc();` 로 첫 번째 slab에 대한 첫 페이지를 할당합니다.
  - 페이지 수 설정: `stable.slabs[i].num_pages = 1;` 로 초기 페이지 수를 설정합니다.
- 락 해제: `release(&stable.lock)` 을 통해 락을 해제합니다.

## ▼ Slab memory allocation

### kmalloc 함수의 구현

```

char *kmalloc(int size){

    //out of range error needs to be handled
    if(size > 2048 || size<=0)
        return 0;
    //choose the byte 8 or 16 .. etc by getting index of slab
    int slabIdx=getslabIdx(nextPowerOf2(size));

```

```

struct slab *s;
s=&stable.slab[slabIdx];
//can't alloc if num of used object is full
if(s->num_used_objects==s->num_objects_per_page*MAX_PAGES_PER_SLAB)
    return 0;
if(stable.slab[0].num_used_objects==stable.slab[0].num_objects_per_page*64)
    return 0;

int startOffset=0;
acquire(&stable.lock);
if((startOffset=checkNewpage(slabIdx)) && s->num_used_objects!=0)
{
    s->page[startOffset/s->num_objects_per_page+1]=kalloc(); //current page +1
}

int bitOffset=setBitmap(slabIdx);
//getPageNumfrom bitmap
s->num_pages=getpageNum(slabIdx);
s->num_free_objects-=1;
s->num_used_objects+=1;

int nowpage=bitOffset/s->num_objects_per_page+1;
int pageOffset=(bitOffset%s->num_objects_per_page)*(1<<(slabIdx+3))*sizeof(char);
memset(s->page[nowpage]+pageOffset,0,size*sizeof(char));
release(&stable.lock);
return s->page[nowpage]+pageOffset;
}

```


## kmalloc()함수에 대한 설명

### 1. 범위 체크

```

if(size > 2048 || size<=0)
    return 0;

```

- 할당 요청된 메모리 크기가 0보다 작거나 2048 바이트보다 크면 유효하지 않은 요청으로 간주하고  을 반환합니다.

### 2. Slab 인덱스 결정

```

int slabIdx = getslabIdx(nextPowerOf2(size));
struct slab *s;
s = &stable.slab[slabIdx];

```



- `nextPowerOf2(size)` 함수를 사용하여 요청된 크기보다 크거나 같은 가장 작은 2의 거듭제곱 값을 구합니다.
- `getslabIdx` 함수로 적절한 slab 인덱스를 얻습니다.
- 해당 slab 인덱스를 통해 slab 포인터 `s` 를 설정합니다.

### 3. Slab 사용 가능 여부 확인

```

if(s->num_used_objects == s->num_objects_per_page * MAX_PAGES_PER_SLAB)
    return 0;
if(stable.slab[0].num_used_objects == stable.slab[0].num_objects_per_page * 64)
    return 0;

```

- 해당 slab에 사용 가능한 객체가 없는 경우  을 반환합니다.
- 첫 번째 slab (`stable.slab[0]`)가 이미 가득 찬 경우도  을 반환합니다.

### 4. 새로운 페이지 할당

```

int startOffset = 0;
acquire(&stable.lock);
if((startOffset = checkNewpage(slabIdx)) && s->num_used_objects != 0)
{

```

```
s->page[startOffset / s->num_objects_per_page + 1] = kalloc(); // current page +1
}
```

- `checkNewpage(slabIdx)` 함수로 새 페이지를 할당해야 하는지 확인합니다.
- 새 페이지가 필요하고 사용된 객체가 있는 경우 `kalloc()` 을 사용하여 새 페이지를 할당합니다.
- 락을 획득하여 멀티스레드 환경에서도 안전하게 처리합니다.

## 5. 비트맵 업데이트 및 페이지 수 갱신

```
int bitOffset = setBitmap(slabIdx);
// getPageNumfrom bitmap
s->num_pages = getpageNum(slabIdx);
s->num_free_objects -= 1;
s->num_used_objects += 1;
```

- `setBitmap(slabIdx)` 함수로 비트맵을 업데이트하여 새로운 객체의 비트를 설정합니다.
- `getpageNum(slabIdx)` 함수로 현재 사용 중인 페이지 수를 갱신합니다.
- 사용 가능한 객체 수를 줄이고, 사용된 객체 수를 증가시킵니다.

## 6. 메모리 초기화 및 포인터 반환

```
int nowpage = bitOffset / s->num_objects_per_page + 1;
int pageOffset = (bitOffset % s->num_objects_per_page) * (1 << (slabIdx + 3)) * sizeof(char);
memset(s->page[nowpage] + pageOffset, 0, size * sizeof(char));
release(&stable.lock);
return s->page[nowpage] + pageOffset;
```

- `bitOffset` 을 이용하여 현재 페이지와 페이지 내 오프셋을 계산합니다.
- `memset` 을 사용하여 할당된 메모리 영역을 `0` 으로 초기화합니다.
- 락을 해제합니다.
- 할당된 메모리의 포인터를 반환합니다.

## ▼ Slab memory deallocation

### kmfree함수의 구현

```
void kmfree(char *addr, int size){
    struct slab *s;
    int slabIdx=getslabIdx(size);
    s=&stable.slab[slabIdx];

    acquire(&stable.lock);
    if(s->num_used_objects==0)
    {
        release(&stable.lock);
        return;
    }

    //set the garbage in slab;
    memset(addr,1,size);
    //bitmap operation
    //get the adress page Number
    int pageNum=0;
    for(int i=1;i<=100;i++)
    {
        if(addr-s->page[i]>=0 && addr-s->page[i]<PGSIZE)
        {
            pageNum=i;
            break;
        }
    }
    //get the offset to make 0 appropriate bitmap
```

```

int offset=0;
offset=((pageNum-1)*s->num_objects_per_page)+(addr-s->page[pageNum])/s->size;

if(clearBitmap(slabIdx,offset))
{
    stable.slab[slabIdx].num_free_objects+=1;
    stable.slab[slabIdx].num_used_objects-=1;
}

//page free
int startOffset=(pageNum-1)*s->num_objects_per_page;
int endOffset=(startOffset+s->num_objects_per_page)-1;
if(pageNum!=1 && checkEmpty(startOffset,endOffset,slabIdx))
{
    kfree(s->page[pageNum]);
    s->num_pages-=1;
}
release(&stable.lock);
// return;
}

```

## kmalloc함수에 대한 설명

### 1. Slab 인덱스 결정 및 Slab 선택

```

struct slab *s;
int slabIdx = getslabIdx(size);
s = &stable.slab[slabIdx];

```

- `getslabIdx(size)` 함수를 사용하여 주어진 크기에 맞는 slab의 인덱스를 얻습니다.
- 인덱스를 이용하여 해당 slab 포인터 `s`를 설정합니다.

### 2. Slab 사용 가능 여부 확인

```

acquire(&stable.lock);
if(s->num_used_objects == 0)
{
    release(&stable.lock);
    return;
}

```

- 락을 획득하여 멀티스레드 환경에서 안전하게 처리합니다.
- 사용된 객체가 없는 경우(즉, 할당된 객체가 없는 경우) 락을 해제하고 함수에서 반환합니다.

### 3. 할당 해제된 메모리 설정

```

// set the garbage in slab
memset(addr, 1, size);

```

- 해제될 메모리 주소 `addr`에 `1` 값을 설정하여 슬랩 내의 데이터를 덮어씁니다. 이는 주로 디버깅 목적으로 사용됩니다. 메모리를 해제했다고 보편됩니다.

### 4. 주소 페이지 번호 확인

```

// get the address page number
int pageNum = 0;
for(int i = 1; i <= 100; i++)
{
    if(addr - s->page[i] >= 0 && addr - s->page[i] < PGSIZE)
    {
        pageNum = i;
        break;
    }
}

```



```

    }
}

```

- 해제될 주소 `addr` 이 속한 페이지 번호를 찾습니다.
- 슬랩의 페이지 배열 `s->page` 에서 `addr` 이 포함된 페이지를 찾아서 `pageNum` 에 설정합니다.

## 5. 비트맵 업데이트

```

// get the offset to make 0 appropriate bitmap
int offset = 0;
offset = ((pageNum - 1) * s->num_objects_per_page) + (addr - s->page[pageNum]) / s->size;

if(clearBitmap(slabIdx, offset))
{
    stable.slab[slabIdx].num_free_objects += 1;
    stable.slab[slabIdx].num_used_objects -= 1;
}

```

- 페이지 번호와 주소로부터 오프셋을 계산합니다.
- `clearBitmap(slabIdx, offset)` 함수를 사용하여 비트맵에서 해당 오프셋의 비트를 클리어합니다.
- 비트맵 클리어가 성공하면 여유 객체 수를 증가시키고 사용된 객체 수를 감소시킵니다.

## 6. 페이지 해제

```

// page free
int startOffset = (pageNum - 1) * s->num_objects_per_page;
int endOffset = (startOffset + s->num_objects_per_page) - 1;
if(pageNum != 1 && checkEmpty(startOffset, endOffset, slabIdx))
{
    kfree(s->page[pageNum]);
    s->num_pages -= 1;
}
release(&stable.lock);

```

- 페이지의 시작 오프셋과 끝 오프셋을 계산합니다.
- 첫 페이지가 아니고( `pageNum != 1` ), 해당 페이지가 비어 있는 경우 `kfree(s->page[pageNum])` 로 페이지를 해제합니다.
- 페이지 수를 감소시킵니다.
- 락을 해제하여 함수에서 반환합니다.

### ▼ Debug 관련 함수

#### slabdump함수의 구현

```

void slabdump(){
    cprintf("__slabdump__\n");

    struct slab *s;

    cprintf("size\tnum_pages\tused_objects\tfree_objects\n");

    for(s = stable.slab; s < &stable.slab[NSLAB]; s++){
        cprintf("%d\t%d\t\t%d\t\t%d\n",
            s->size, s->num_pages, s->num_used_objects, s->num_free_objects);

        // Print the bitmap for each slab
        cprintf("Bitmap: ");
        for(int i=0;i<60;i++)
        {
            for(int j=7;j>=0;j--){
                cprintf("%d",get_bit(s->bitmap[i],j));
            }
        }
    }
}

```

```

        cprintf(" ");
    }
    cprintf("\n");
}
}

```

이 함수는 slab의 메모리 상태를 확인하며 디버깅을 하기 위한 용도입니다.

본 함수에서는 사전에 구현된 cprintf 함수를 이용하여 slab 구조체의 `size`, `num_pages`, `num_used_objects`, `num_free_objects` 필드 값을 바이트 단위로 출력합니다.

## ▼ 4. Test 및 구현

### ▼ linux환경에서 test를 하는 법

1. 터미널에서 다음 명령어를 실행합니다.

```
git clone https://github.com/SSUminiOS/A3teamOS/tree/main
```

2. 터미널에서 makefile이 있는 디렉토리인 src에 접근을 합니다.

```
cd src
```

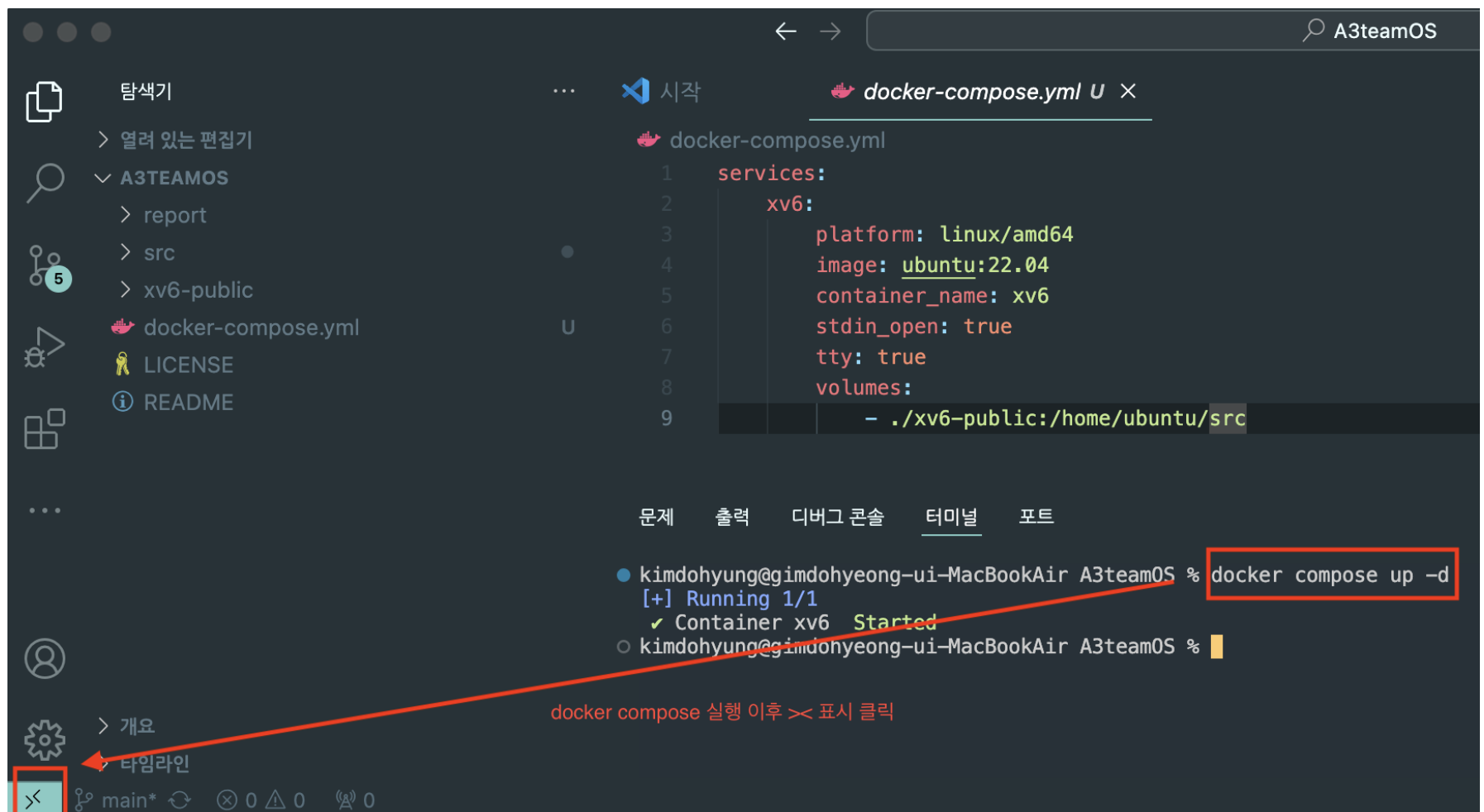
3. 터미널에서 명령어로 make qemu-nox를 입력합니다.

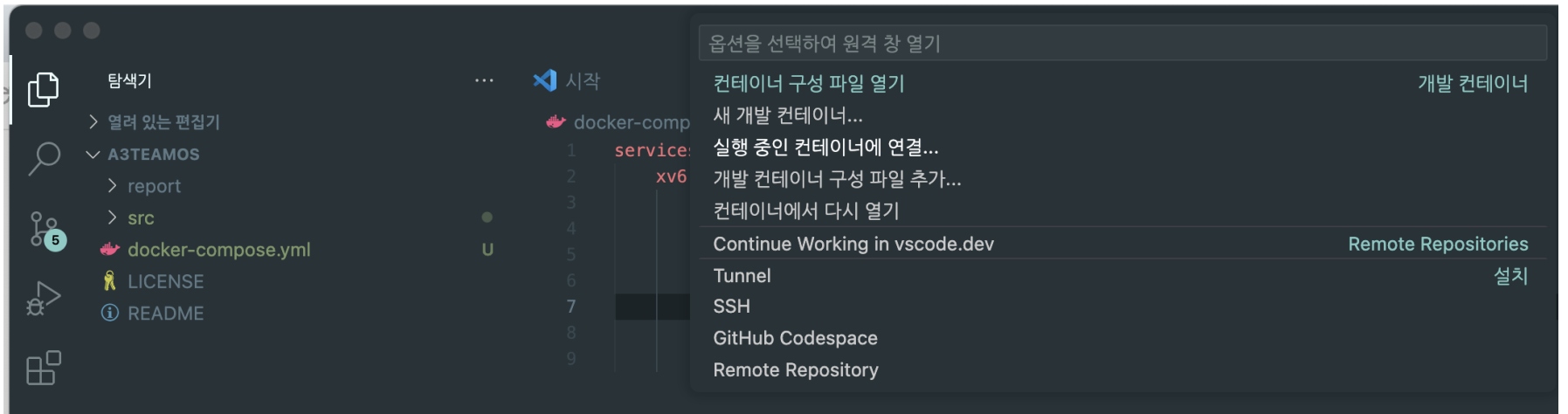
```
make qemu-nox
```

4. xv6를 실행 후 slabtest를 입력하여 출력 결과를 확인합니다.

```
slabtest
```

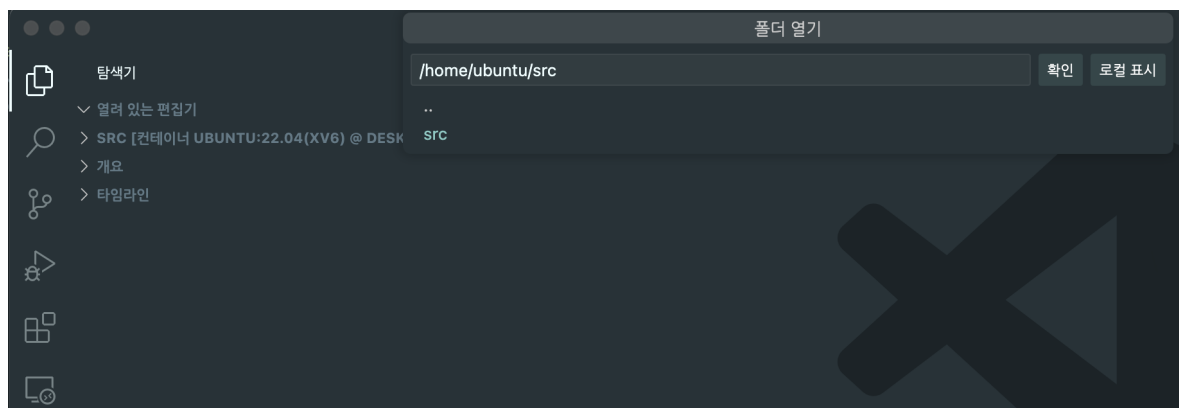
### ▼ docker환경에서 test를 하는 법



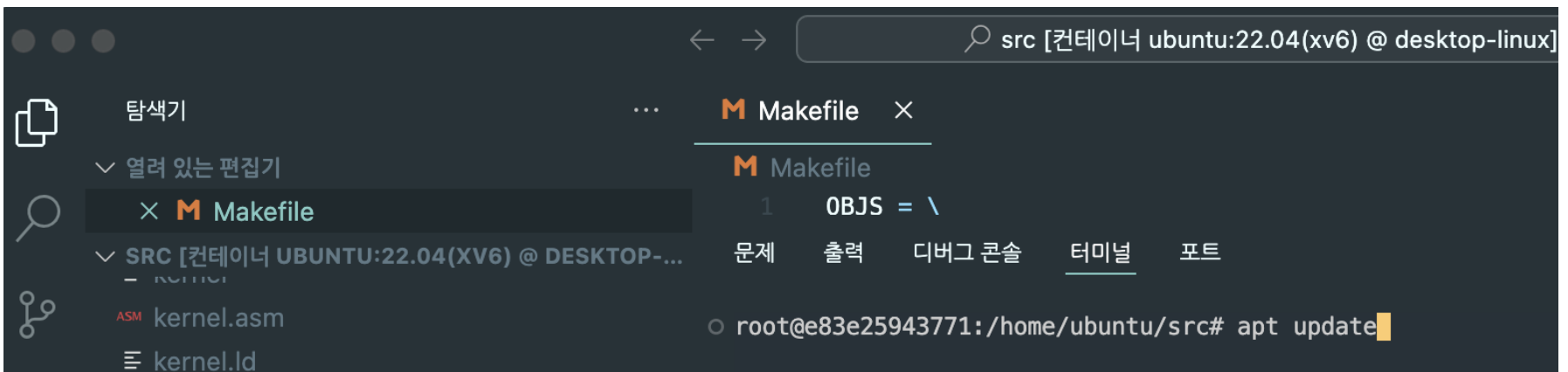


git clone 이후, vscode에서 docker-compose 파일을 실행 `docker compose up -d`

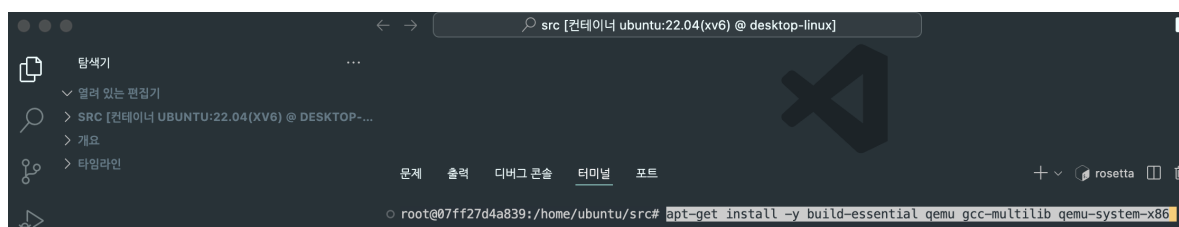
그리고 vscode 좌하단에 ><를 클릭하여 실행 중인 컨테이너에 연결을 누르고 /xv6에 연결



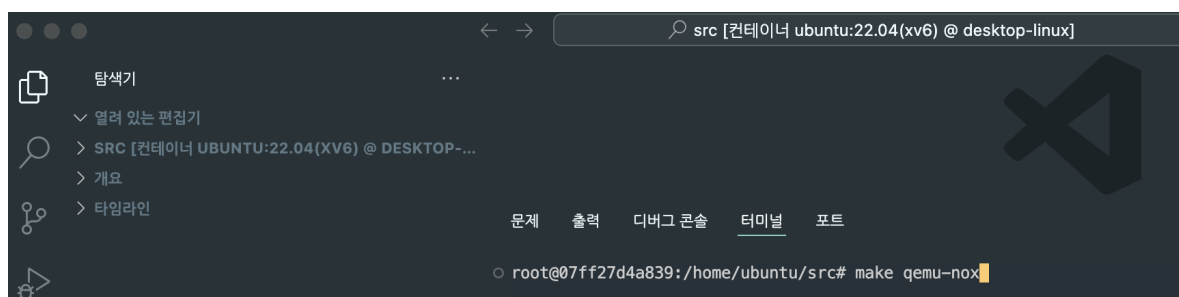
다음 vscode 파일 열기를 통해 `/home/ubuntu/src` 경로에 접속

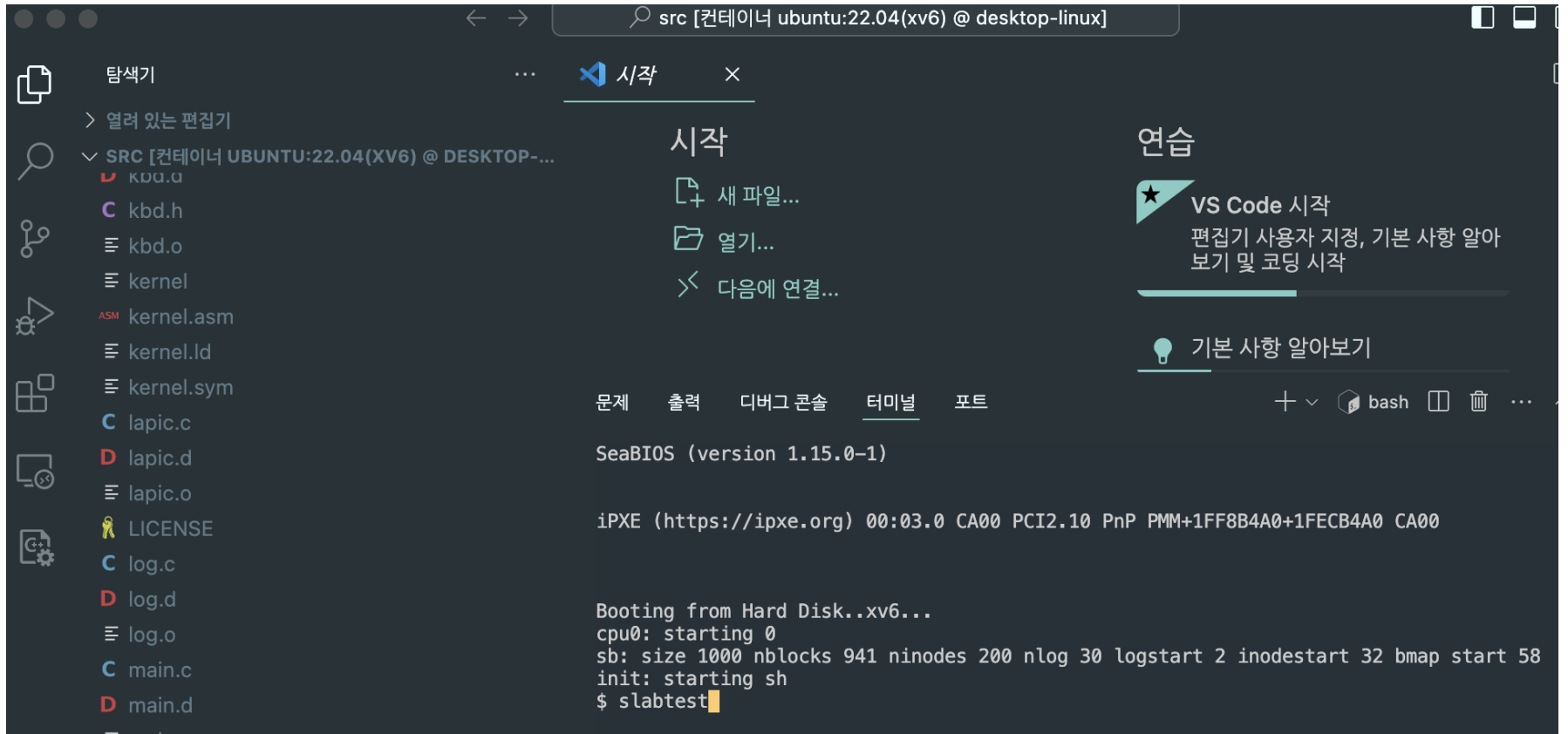


`apt-update` 명령어 실행



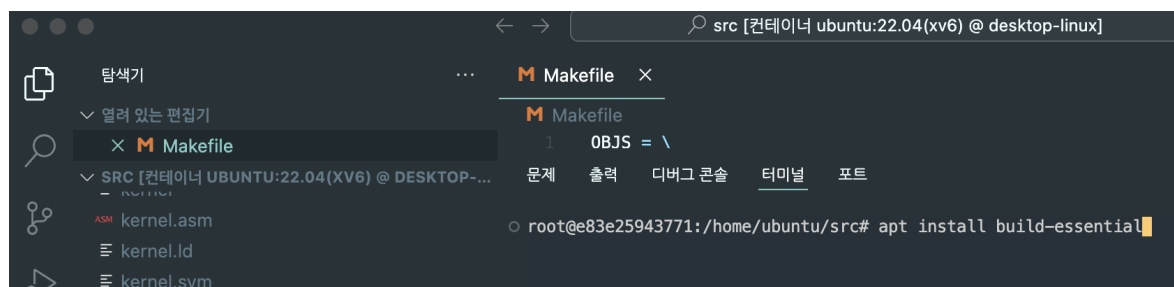
`apt-get install -y build-essential qemu gcc-multilib qemu-system-x86` 명령어 실행





`make qemu-nox`로 `build` 하고 `slabtest` 실행 시 테스트 결과 확인 가능

▼ 만약 `make qemu-nox`에서 `make: *** No rule to make target gnu/9/include/stdbool.h', needed by 'slab.o'. Stop.` 오류가 발생한 경우



`apt install build-essential` 명령어 실행

## ▼ Test program에 대한 설명

### ▼ Test case 1-6

#### Test case 1

```
/* TEST1: Single slab alloc */
cprintf("==== TEST1 =====\n");
start = counter;
t[0][0] = (int *)kmalloc(TESTSIZE);
*(t[0][0]) = counter;
counter++;
slabdump();
cprintf((*t[0][0]) == start && numobj_slab(TESTSLABID) == 1) ? "OK\n" : "WRONG\n");
kmfree((char *)t[0][0], TESTSIZE);
slabdump();
```

할당된 메모리 블록의 유효성을 검증하기 위해 사용되며 지정된 크기의 메모리를 할당받아 변수에 저장됩니다. 할당된 메모리 블록에 현재 카운터 값을 저장합니다. 이는 할당된 메모리가 올바르게 초기화되었는지 확인하기 위함이며, 카운터 값을 증가시키며 `slab`의 상태를 출력합니다.

메모리 블록이 올바르게 초기화 되었다면 `"OK"`를 출력합니다.

#### Test case 2

```
/* TEST2: Single slab alloc: the size not equal to a power of 2. */
cprintf("==== TEST2 =====\n");
start = counter;
t[0][0] = (int *)kmalloc(TESTSIZE - 10);
*(t[0][0]) = counter;
```

```

slabdump();
counter++;
cprintf((*t[0][0]) == start && numobj_slab(TESTSLABID) == 1) ? "OK\n" : "WRONG\n");
kfree((char *)t[0][0], TESTSIZE);
slabdump();

```

슬랩 할당자를 이용한 메모리 관리에서, 할당 크기가 2의 제곱수가 아닌 경우에도 올바르게 동작하는지를 확인합니다.

테스트 시작을 알리는 메시지를 출력합니다. 현재 카운터 값을 저장합니다. 이는 할당된 메모리 블록의 유효성을 검증하기 위해 사용됩니다.

**TESTSIZE** 보다 10byte 작은 크기의 메모리를 할당받아 변수에 저장합니다. 할당된 메모리 블록에 현재 카운터 값을 저장합니다. 이는 할당된 메모리가 올바르게 초기화되었는지 확인하기 위함입니다. 현재 **slab**의 상태를 출력합니다. 이는 현재 메모리 할당 상태를 확인하고 디버깅을 돕기 위함입니다. 카운터 값을 증가시킵니다. 다음 할당 시 고유한 값을 배정하기 위해서입니다. 할당된 메모리 블록이 올바르게 초기화되었는지와 slab에 객체가 하나만 존재하는지를 확인합니다. 조건이 만족되면 **"OK"**를, 그렇지 않으면 **"WRONG"**을 출력합니다. 이전에 할당받은 메모리를 해제합니다. 여기서 해제할 때는 원래 **TESTSIZE** 크기를 사용합니다. 다시 slab의 상태를 출력하여 메모리 해제가 올바르게 이루어졌는지 확인합니다.

### Test case 3

```

/* TEST3: Multiple slabs alloc */
cprintf("==== TEST3 =====\n");
start = counter;
for (int i = 0; i < NSLAB; i++)
{
    int slabsize = 1 << (i + 3);
    t[i][0] = (int *)kmalloc(slabsize);
    for (int j = 0; j < slabsize / sizeof(int); j++)
    {
        memmove(t[i][0] + j, &counter, sizeof(int));
        counter++;
    }
}

// CHECK
pass = 1;
for (int i = 0; i < NSLAB; i++)
{
    int slabsize = 1 << (i + 3);
    for (int j = 0; j < slabsize / sizeof(int); j++)
    {
        // cprintf("%d, %d, %d, %d\n", i, j, *(t[i][0]+j), start);      //YOU MAY USE THIS
        if (*(t[i][0] + j) != start)
        {
            pass = 0;
            break;
        }
        start++;
    }
}
slabdump();
cprintf(pass ? "OK\n" : "WRONG\n");
for (int i = 0; i < NSLAB; i++)
{
    int slabsize = 1 << (i + 3);
    kfree((char *)t[i][0], slabsize);
}
slabdump();

```

테스트 시작을 알리는 메시지를 출력합니다. 현재 카운터 값을 저장합니다. 이는 할당된 메모리 블록의 유효성을 검증하기 위해 사용됩니다. 여러 개의 **slab**을 순차적으로 할당합니다. 각 **slab**의 크기는 8byte ( $2^3$ )부터 시작하여 이후 슬랩은 이전 **slab**의 두 배 크기입니다. 할당된 각 slab에 대해, **slab** 크기만큼의 정수 배열을 생성하고, 배열의 각 요소에 현재 카운터 값을 저장합니다. 카운터 값은 각 요소마다 증가시킵니다.

모든 **slab** 할당 및 데이터 저장이 완료된 후, 저장된 데이터가 올바르게 보존되었는지를 확인합니다. 각 **slab**의 모든 요소를 검사하여, 요소의 값이 기대한 값과 일치하는지 확인합니다. 모든 값이 일치하면 **pass** 변수를 1로 설정하고, 그렇지 않으면 0으로 설정합니다. 현재 **slab**의 상태를 출력하여, 할당된 메모리의 상태를 확인합니다. 데이터 검사가 통과되면 **"OK"**를, 그렇지 않으면 **"WRONG"**을 출력합니다.

모든 **slab** 할당이 완료된 후, 할당된 memory를 해제합니다. 각 **slab**의 크기는 할당 시의 크기와 동일합니다. 마지막으로 **slab**의 상태를 다시 출력하여, memory 해제가 올바르게 이루어졌는지 확인합니다.

#### Test case 4

```
/* TEST4: Multiple slabs alloc2 */
cprintf("==== TEST4 =====\n");
start = counter;
for (int i = 0; i < NSLAB; i++)
{
    int slabsize = 1 << (i + 3);
    // cprintf("slabsize:%d\n", slabsize);
    for (int j = 0; j < MAXTEST; j++)
    {
        t[i][j] = (int *)kmalloc(slabsize);
        // cprintf("adress: %p\n", (int*)t[i][j]);
        for (int k = 0; k < slabsize / sizeof(int); k++)
        {
            // slabdump();
            memmove(t[i][j] + k, &counter, sizeof(int));
            counter++;
        }
    }
    slabdump();
    // CHECK
    pass = 1;
    for (int i = 0; i < NSLAB; i++)
    {
        int slabsize = 1 << (i + 3);
        for (int j = 0; j < MAXTEST; j++)
        {
            for (int k = 0; k < slabsize / sizeof(int); k++)
            {
                // cprintf("%d, %d, %d, %d, %d\n", i, j, k, *(t[i][j]+k), start);
                if (*(t[i][j] + k) != start)
                {
                    pass = 0;
                    break;
                }
                start++;
            }
        }
    }
    cprintf(pass ? "OK\n" : "WRONG\n");
    // slabdump();
    for (int i = 0; i < NSLAB; i++)
    {
        int slabsize = 1 << (i + 3);
        // cprintf("slabsize:%d\n", slabsize);
        for (int j = 0; j < MAXTEST; j++)
        {
            kfree((char *)t[i][j], slabsize);
            // slabdump();
        }
    }
    slabdump();
}
```

`slab` 크기를 점진적으로 증가시키면서 메모리를 할당하고, 각 할당된 메모리 블록에 `counter` 값을 복사합니다. `slabsize` 는 8부터 시작하여 각 슬랩마다 크기가 2배씩 증가합니다. `kmalloc` 함수를 통해 메모리를 할당하고, `memmove` 함수를 사용하여 해당 메모리 블록에 `counter` 값을 복사합니다. 이 과정에서 `counter` 는 계속 증가합니다.

각 메모리 블록의 값이 초기 `start` 값과 일치하는지 확인하고, 만약 일치하지 않으면 `pass` 변수를 0으로 설정합니다. 모든 검증이 성공하면 `"OK"` 를 출력하고, 실패하면 `"WRONG"` 을 출력합니다.

그 후 저장된 값이 올바르게 되었는지 검증하고 메모리 할당을 해제합니다.

### Test case 5

```
/* TEST5: ALLOC MORE THAN 100 PAGES */
cprintf("==== TEST5 =====\n");
start = counter;
for (int j = 0; j < MAXTEST; j++)
{
    t[0][j] = (int *)kmalloc(TESTSIZE);
    // cprintf("adress: %p", (int*)t[0][j]);
    for (int k = 0; k < TESTSIZE / sizeof(int); k++)
    {
        // slabdump();
        memmove(t[0][j] + k, &counter, sizeof(int));
        counter++;
    }
}
tmp = (int *)kmalloc(TESTSIZE);
cprintf((!tmp && numobj_slab(TESTSLABID) == MAXTEST) ? "OK\n" : "WRONG\n");
slabdump();
```

`kmalloc(TESTSIZE)` 함수를 통해 메모리를 할당하며, `memmove` 함수를 사용하여 할당된 메모리 블록에 `counter` 값을 복사합니다. 이 과정에서 `counter` 는 계속 증가합니다. 또한, 추가적으로 메모리를 할당하며 만약 메모리 할당이 실패하고(`!tmp`), 현재 슬랩에 할당된 객체 수 (`numobj_slab(TESTSLABID)`)가 `MAXTEST` 와 같다면 `"OK"` 를 출력합니다. 그렇지 않으면 `"WRONG"` 을 출력합니다. `slabdump` 함수는 현재 슬랩 메모리 상태를 출력합니다.

### Test case 6

```
/* TEST6: ALLOC AFTER FREE */
cprintf("==== TEST6 =====\n");
for (int j = 0; j < MAXTEST; j++)
{
    kmfree((char *)t[0][j], TESTSIZE);
}
slabdump();
start = counter;
for (int j = 0; j < MAXTEST; j++)
{
    t[0][j] = (int *)kmalloc(TESTSIZE);
    for (int k = 0; k < TESTSIZE / sizeof(int); k++)
    {
        memmove(t[0][j] + k, &counter, sizeof(int));
        counter++;
    }
}
slabdump();
// CHECK
pass = 1;
for (int j = 0; j < MAXTEST; j++)
{
    for (int k = 0; k < TESTSIZE / sizeof(int); k++)
    {
        if (*(t[0][j] + k) != start)
        {
            pass = 0;
            break;
        }
        start++;
    }
}
cprintf(pass ? "OK\n" : "WRONG\n");
for (int j = 0; j < MAXTEST; j++)
{
    kmfree((char *)t[0][j], TESTSIZE);
    // slabdump();
}
```

```

    }
    slabdump();

```

이 부분에서는 이전에 할당한 `MAXTEST` 개의 메모리 블록을 모두 해제합니다. `kmfree` 함수를 사용하여 각 블록을 해제하며, `slabdump` 함수를 통해 현재 슬랩 메모리 상태를 출력합니다.

또한, 해제된 메모리를 다시 할당합니다. `kmalloc(TESTSIZE)` 함수를 통해 메모리를 할당하며, 다시 `memmove` 함수를 사용하여 할당된 메모리 블록에 `counter` 값을 복사합니다. 이 과정에서 `counter` 는 계속 증가합니다. 이후 `slabdump` 함수를 통해 현재 슬랩 메모리 상태를 출력합니다.

또한, 재할당된 메모리 블록의 값을 검증하며 `"OK"` 또는 `"WRONG"` 을 출력합니다.

## ▼ Test case 7-8

### Test case 7

```

// alloc and free and alloc
cprintf("==== TEST7 =====\n");
for (int j = 0; j < 16; j++)
{
    t[0][j] = (int *)kmalloc(TESTSIZE2);
    for (int k = 0; k < TESTSIZE2 / sizeof(int); k++)
    {
        memmove(t[0][j] + k, &counter, sizeof(int));
        counter++;
    }
}
slabdump();
for (int j = 10; j < 12; j++)
{
    kmfree((char *)t[0][j], TESTSIZE2);
}
for (int j = 13; j < 15; j++)
{
    kmfree((char *)t[0][j], TESTSIZE2);
}
slabdump();
for (int j = 10; j < 12; j++)
{
    t[0][j] = (int *)kmalloc(TESTSIZE2);
    for (int k = 0; k < TESTSIZE2 / sizeof(int); k++)
    {
        memmove(t[0][j] + k, &counter, sizeof(int));
        counter++;
    }
}
slabdump();
for (int j = 8; j < 16; j++)
{
    kmfree((char *)t[0][j], TESTSIZE2);
}
slabdump();
for (int j = 0; j < 8; j++)
{
    kmfree((char *)t[0][j], TESTSIZE2);
}
slabdump();

```

`kmalloc` 함수를 사용하여 16개의 메모리 블록을 할당합니다. 각 블록의 크기는 `TESTSIZE2` 이며, 할당된 메모리 블록은 `t[0][j]` 에 저장됩니다. 각 블록은 `counter` 값을 사용하여 초기화됩니다. 초기화 후 현재 메모리 상태를 `slabdump` 함수를 통해 출력합니다.

특정 인덱스(10, 11, 13, 14)의 메모리 블록을 해제합니다. `kmfree` 함수는 메모리 블록을 해제하는 역할을 합니다. 해제 후 현재 메모리 상태를 `slabdump` 함수를 통해 출력합니다.

이전에 해제된 인덱스(10, 11)의 메모리 블록을 다시 할당하고 초기화합니다. 재할당 후 현재 메모리 상태를 `slabdump` 함수를 통해 출력합니다.

마지막으로, 추가로 메모리 블록을 해제합니다. 첫 번째 루프에서는 인덱스 8부터 15까지의 블록을 해제하고, 두 번째 루프에서는 index 0부터 7까지의 블록을 해제합니다. 각 해제 후 현재 메모리 상태를 `slabdump` 함수를 통해 출력합니다.



Test case 8

```
cprintf("==== TEST8 =====\n");
for (int j = 0; j < 24; j++)
{
    t[0][j] = (int *)kmalloc(TESTSIZE2);
    for (int k = 0; k < TESTSIZE2 / sizeof(int); k++)
    {
        memmove(t[0][j] + k, &counter, sizeof(int));
        counter++;
    }
}
slabdump();
for (int j = 8; j < 16; j++)
{
    kmfree((char *)t[0][j], TESTSIZE2);
}
slabdump();
for (int j = 8; j < 16; j++)
{
    t[0][j] = (int *)kmalloc(TESTSIZE2);
    for (int k = 0; k < TESTSIZE2 / sizeof(int); k++)
    {
        memmove(t[0][j] + k, &counter, sizeof(int));
        counter++;
    }
}
slabdump();
for (int j = 0; j < 24; j++)
{
    kmfree((char *)t[0][j], TESTSIZE2);
    slabdump();
}
slabdump();
```

24개의 메모리 블록을 할당하고, 각 블록을 초기화합니다. `kmalloc` 함수를 사용하여 `TESTSIZE2` 크기의 메모리를 할당하고, 메모리 블록을 정수 값으로 초기화하며 각 정수는 증가합니다. 그리고 루프에서 8~15번 까지의 메모리 블록을 다시 할당하고 초기화하며 `slabdump` 함수로 호출합니다. 그리고 메모리를 해제하고 상태를 확인합니다.

특정 패턴의 메모리 할당 및 해제를 통해 메모리 관리 시스템의 동작을 테스트합니다. 각 단계에서 메모리 상태를 확인함으로써, 메모리 할당자 및 해제자의 효율성과 안정성을 평가할 수 있습니다. `slabdump` 함수 호출을 통해 메모리 상태를 모니터링하고, 메모리 누수 및 단편화를 분석할 수 있습니다.

▼ Test 실행결과

Test1 실행결과

```
==== SLAB TEST ====
==== TEST1 =====
slabdump
size  num_pages  used_objects  free_objects
8      1           0       32768
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
16      1           0       25600
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
32      1           0       12800
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
64      1           0        6400
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
128     1           0        3200
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
256     1           0        1600
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
512     1           0         800
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
1024    1           0         400
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
2048    1           1          199
Bitmap: 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
OK
slabdump
size  num_pages  used_objects  free_objects
8      1           0       32768
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
16      1           0       25600
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
32      1           0       12800
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
64      1           0        6400
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
128     1           0        3200
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
256     1           0        1600
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
512     1           0         800
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
1024    1           0         400
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
2048    1           0          200
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Test 1에서는 실행결과 2048byte 크기의 single slab이 할당 된후 정상적으로 해제가 되었습니다.

2048 byte slab의 비트맵이 1에서 0으로 변했고, used\_objects 수와 free\_objects 수가 메모리 할당 전 후로 올바르게 출력되고 있습니다.

Test2 실행결과

```
==== TEST2 =====
_slabdump_
size  num_pages      used_objects    free_objects
8      1              0             32768
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
16     1              0             25600
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
32     1              0             12800
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
64     1              0             6400
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
128    1              0             3200
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
256    1              0             1600
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
512    1              0             800
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
1024   1              0             400
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
2048   1              1             199
Bitmap: 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
OK
_slabdump_
size  num_pages      used_objects    free_objects
8      1              0             32768
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
16     1              0             25600
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
32     1              0             12800
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
64     1              0             6400
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
128    1              0             3200
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
256    1              0             1600
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
512    1              0             800
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
1024   1              0             400
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
2048   1              0             200
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Test 2에서는 파편화를 확인하기위해 Test 2에서는 2의 배수가 아닌 수만큼 slab을 할당하는 작업을 진행하였습니다. 이 테스트에서는 2038바이트 (TESTSIZE - 10) 크기의 슬랩을 할당합니다. 할당이 성공했는지 확인하기 위해 할당된 메모리에 올바른 값이 들어 있는지와 해당 슬랩에 있는 객체 수를 확인하고, 의도한 대로 메모리가 할당 되었음을 확인할 수 있었습니다.

Test3 실행결과

```
==== TEST3 =====
_slabdump_
size  num_pages      used_objects    free_objects
8      1              1             32767
Bitmap: 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
16     1              1             25599
Bitmap: 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
32     1              1             12799
Bitmap: 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
64     1              1             6399
Bitmap: 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
128    1              1             3199
Bitmap: 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
256    1              1             1599
Bitmap: 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
512    1              1             799
Bitmap: 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
1024   1              1             399
Bitmap: 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
2048   1              1             199
Bitmap: 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
OK
_slabdump_
size  num_pages      used_objects    free_objects
8      1              0             32768
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
16     1              0             25600
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
32     1              0             12800
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
64     1              0             6400
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
128    1              0             3200
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
256    1              0             1600
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
512    1              0             800
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
1024   1              0             400
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
2048   1              0             200
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Test 3에서는 slab 할당기가 정의한 각 크기 클래스에 대해 하나의 slab 할당합니다. 각 할당된 slab의 크기는 8바이트부터 최대 크기까지 다양한 크기를 테스트 해보았습니다. 각 슬랩을 정수로 채운 후, 올바른 값이 들어 있는지 확인합니다. 확인 후, 모든 할당된 슬랩을 해제합니다.

Test4 실행결과

[illegible]

Test 4에서는 Test 3와 유사하게 다양한 크기의 slab을 할당합니다. 각 크기마다 최대치인 200개를 할당하며, 예상된 값이 있는지 확인하는 테스트를 진행합니다.

올바른 메모리가 할당된 것을 확인한 후 할당된 slab을 해제합니다.

## Test5 실행결과

[illegible]

Test 5에서는 2048byte 크기의 슬랩을 200개 최대 할당하여 총 100 페이지를 초과하는 메모리를 할당합니다. 할당 제한에 도달한 후 추가 할당이 실패하고 **NULL** 을 반환하는지 확인합니다. 확인 후, 모든 할당된 슬랩을 해제합니다.

## Test6 실행결과

[illegible]



Test 6에서는 할당 해제된 메모리 영역에 재할당을 진행하였습니다. 그 후 각 주소에 올바른 값이 할당 되었는지 확인 후, slab을 해제합니다.

## Test7 실행결과

[illegible]

해당 소스코드는 512byte 크기의 객체 16개를 할당합니다. 중간에 의미로 특정 객체(10~11, 13~14)를 해제한 후 다시 할당합니다. slab 할당기가 부분 해제 및 부분 재할당을 올바르게 처리할 수 있는지 확인하고, 모든 slab을 해제하는 모습을 확인할 수 있었습니다.

## Test8 실행결과

[illegible]

```
__slabdump__
size      num_pages      used_objects      free_objects
8         1              0              32768
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
16        1              0              25600
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
32        1              0              12800
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
64        1              0              6400
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
128       1              0              3200
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
256       1              0              1600
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
512       3              24              776
Bitmap: 11111111 11111111 11111111 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
1024      1              0              400
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
2048      1              0              200
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
__slabdump__
size      num_pages      used_objects      free_objects
8         1              0              32768
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
16        1              0              25600
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
32        1              0              12800
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
64        1              0              6400
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
128       1              0              3200
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
256       1              0              1600
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
512       1              0              800
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
1024      1              0              400
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
2048      1              0              200
Bitmap: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Test 8에서는 TEST7과 유사하지만, 512bytes 크기의 slab 24개를 할당하고 중간에 있는 페이지 전체를 해제한 후 다시 할당합니다. 다시 원상태로 할당을 한 뒤에 메모리를 해제합니다. 이 테스트는 한 page전체를 해제하고 다시 페이지를 할 당 할 수 있는지를 검증합니다.

▼ 5. Trouble shooting

Xv6 시스템 콜 적용 실패 문제 해결

**Problem:** 새로운 시스템 콜 `sys_ps` 를 추가했으나 유저 프로그램에서 `ps()` 함수를 호출했을 때, 예상한 프로세스 리스트가 출력되지 않음.

**Solution:** 문제의 원인을 분석하고, 각 선언부의 의미를 이해한 후, 수정 사항을 적용.

각 선언부의 의미와 역할

파일명	의미 및 역할
<code>syscall.h</code>	시스템 콜 번호를 정의하여 커널이 새로운 시스템 콜을 인식하게 함
<code>syscall.c</code>	새로운 시스템 콜 핸들러 함수의 프로토타입을 선언하여 참조 가능하게 함
<code>sysproc.c</code>	새로운 시스템 콜 핸들러 함수를 구현하여 시스템 콜의 실제 동작을 정의
<code>syscall.c</code>	시스템 콜 테이블에 핸들러 함수를 추가하여 시스템 콜이 호출되었을 때 연결
<code>user.h</code>	유저 프로그램에서 시스템 콜을 호출할 수 있는 함수 프로토타입을 선언
<code>usys.S</code>	유저 프로그램에서 시스템 콜을 호출할 수 있는 엔트리 포인트를 추가
<code>ulib.c</code>	유저 프로그램에서 시스템 콜을 호출하는 래퍼 함수를 구현

해결 방법 적용

1. `syscall.h` 파일 수정:

- 시스템 콜 번호 정의.

```
#define SYS_ps 23
```

2. `syscall.c` 파일 수정:

- 시스템 콜 핸들러 함수의 프로토타입 선언.

```
extern int sys_ps(void);
```

3. `sysproc.c` 파일 수정:

- 시스템 콜 핸들러 함수 구현.

```
int sys_ps(void) {
    // 프로세스 정보를 출력하는 코드 구현
    return 0;
}
```

4. `syscall.c` 파일 수정:

- 시스템 콜 테이블에 핸들러 함수 추가.

```
[SYS_ps]    sys_ps,
```

#### 5. `user.h` 파일 수정:

- 유저 프로그램에서 시스템 콜을 호출할 수 있는 함수 프로토타입 선언.

```
int ps(void);
```

#### 6. `usys.S` 파일 수정:

- 유저 프로그램에서 시스템 콜을 호출할 수 있는 엔트리 포인트 추가.

```
SYSCALL(ps)
```

#### 7. `ulib.c` 파일 수정:

- 유저 프로그램에서 시스템 콜을 호출하는 래퍼 함수 구현.

```
int ps(void) {
    return syscall(SYS_ps);
}
```

## 최종 확인

모든 수정 후, Xv6를 재빌드하고 유저 프로그램에서 `ps()` 함수를 호출하여 프로세스 리스트가 정상적으로 출력되는지 확인. 추가되는 모든 시스템콜에서 적용하여 문제가 발생하지 않도록 대비

## Test5에서 발생한 문제

```
$ slabtest
==== SLAB TEST ====
==== TEST1 ====
OK
==== TEST2 ====
OK
==== TEST3 ====
OK
==== TEST4 ====
OK
==== TEST5 ====
unexpected trap 14 from cpu 0 eip 80102237 (cr2=0x1010101)
lapicid 0: panic: trap
8010531b 80105064 80106a90 80107026 801071e8 801042da 8010516d 80105064 0 0
```

Test5에서 메모리 할당 중 실패를 했다.

slabdump함수로 출력을 했을 때 어느 부분에서 메모리 할당이 실패를 하는지 찾기 어려워서 다음과 같이 디버깅을 해보았다.

현재 kmalloc과 kmfree를 할 때 현재 할당하고 해제하는 메모리 주소를 출력하는 함수를 만들어서 다음과 같이 page내부의 주소를 출력을 해보았다.

```
address is 8dff2000
address is 8dff2200
address is 8dff2400
address is 8dff2600
address is 8dff2800
address is 8dff2a00
address is 8dff2c00
address is 8dff2e00
address is 8de59000
address is 8de59200
address is 8de59400
address is 8de59600
address is 8de59800
address is 8de59a00
address is 8de59c00
address is 8de59e00
address is 8de5a000
address is 8de5a200
address is 8de5a400
address is 8de5a600
address is 8de5a800
address is 8de5aa00
address is 8de5ac00
address is 8de5ae00
```

주소 출력을 통해 Test4에서 각 slab마다 200개의 객체를 할당을 하고 메모리 해제를 하는 과정에서 오류가 발생함을 확인을 하였다. 따라서 kmfree함수의 일부를 수정을 하였다. 원래 memory allocation하는 부분에서만 bitmap을 사용을 하였으나, kmfree함수에서도 bitmap을 사용을 하였다. kmfree 함수에는 kmalloc함수와 다르게 address를 인자로 받기 때문에 addr값을 통하여 pageNum과 offset을 구하여 비트맵에서도 올바른 위치에서 비트 연산을 할 수 있게 코드를 수정하였다.

Test6에서 발생한 문제

```
==== TEST5 ====
OK
__slabdump__
size    num_pages    used_objects    free_objects
8        1            0               32768
16       1            0               25600
32       1            0               12800
64       1            0               6400
128      1            0               3200
256      1            0               1600
512      1            0               800
1024     1            0               400
2048     100          200             0
==== TEST6 ====
__slabdump__
size    num_pages    used_objects    free_objects
8        1            0               32768
16       1            0               25600
32       1            0               12800
64       1            0               6400
128      1            0               3200
256      1            0               1600
512      1            0               800
1024     1            0               400
2048     100          199             1
0 1 1
lapicid 0: panic: kfree
```

test6에서 발생한 문제도 test5에서 발생한 것과 비슷하게 메모리 해제가 잘 안된 상태에서 할당을 하려고 해서 문제가 되었다. Test5에서 사용한 해결방법으로 위 문제를 해결하였다.

test8에서 발생한 문제

test8에서는 메모리를 할당한 다음에 중간에 있는 page한 개를 비우고 다시 할당을 한다. 이때 해제하는 Memory adress를 잘못 받아와서 잘못된 메모리 주소를 해제하는 경우가 발생을 하였다.



따라서 다음 함수 CheckEmpty함수를 만들어서 문제를 해결하였다.

```
bool checkEmpty(int startOffset,int endOffset,int slabIdx)
{
    struct slab *s;
    s=&stable.slab[slabIdx];
    bool empty=true;
    int startRow=getRow(startOffset);
    int startColumn=getColumn(startOffset);
    int endRow=getRow(endOffset);
    int endColumn=getColumn(endOffset);
    // cprintf("startOffset: %d, endOffset %d\n",startOffset,endOffset);
    // cprintf("startRow:%d endRow:%d\n",startRow,endRow);
    for(int i=startRow;i<=endRow;i++)
    {
        for(int j=startColumn;j<=endColumn;j++)
        {
            if(get_bit(s->bitmap[i],j))
            {
                empty=false;
                break;
            }
        }
        if(empty==false)
            break;
    }
    return empty;
}
```

checkEmpty함수는 페이지에 객체가 하나라도 남아있으면 Empty하지 않다고 판별을 한다. 이때 메모리에 직접 접근하지 않고 bitmap을 확인함으로써 비트맵 계산이 잘 된 상태에서라면 페이지가 비었는지 알 수 있다.

## 적절한 pageNum을 가져오는 문제

slab별로 page안에 들어갈 수 있는 객체 수가 달랐다. 비트맵의 width가 8인데 1024byte slab과 2048byte slab은 페이지 안에 들어는 객체 수가 8개 미만 이어서 비트맵으로 page의 번호를 확인하는데 추가적인 작업이 필요했다. 따라서 현재 page의 개수를 반환하는 함수인 getPageNum을 만들 때 slabIndex별로 다르게 구현을 했다. 2048byte와 1024byte slab은 3중 for문으로 구현을 하였다.

```
int getPageNum(int slabIdx)
{
    struct slab *s;
    s=&stable.slab[slabIdx];
    int page=0;
    //size 2048 - 1024
    if(slabIdx>=7)
    {
        for(int i=0;i<PGSIZE;i++)
        {
            for(int j=0;j<=7;j+=(PGSIZE/s->size))
            {
                for(int k=j;k<j+(PGSIZE/s->size);k++)
                {
                    if(get_bit(s->bitmap[i],k))
                    {
                        page++;
                        break;
                    }
                }
            }
        }
    }
    //size 8 - 512
    else
```



```

{
    for(int i=0;i<PGSIZE;i+=(512/s->size))
    {
        for(int j=i;j<i+(512/s->size);j++)
        {
            if(s->bitmap[j]!=0x00)
            {
                page++;
                break;
            }
        }
    }
    return page;
}

```

### ▼ 6. 프로젝트 일지

이름	허성현(조장), 윤재선, 김도형, 권정태
모임 일자	수행내용
4/14	<p>프로젝트 초기 논의 및 역할 분담</p> <p><b>허성현:</b> 메시지 인터럽트 처리 정책 구체화</p> <p><b>윤재선:</b> slab allocator 설계</p> <p><b>김도형:</b> 소켓 프로그래밍 및 페이징 교체 알고리즘 구상</p> <p><b>권정태:</b> Bash shell 구현 및 RTOS 활용 검토</p>
4/24	<p>운영체제 팀 프로젝트 아이디어 구체화 회의</p> <p><b>허성현:</b> 메시지 인터럽트 처리 정책 세부 구현</p> <p><b>윤재선:</b> slab allocator 구현 계획 수립</p> <p><b>김도형:</b> 운영체제 기본 기능 구현 계획</p> <p><b>권정태:</b> Bash shell 및 간이 OS/Kernel 구현 방향 설정</p>
5/1	<p>프로젝트 업무 분담 및 아이디어 선정</p> <p><b>허성현:</b> 메시지 구조 및 인터럽트 로직 구체화</p> <p><b>윤재선:</b> slab allocator 세부 구현 착수</p> <p><b>김도형:</b> 운영체제 기본 커널 동작 구현</p> <p><b>권정태:</b> 기본 운영체제 기능 추가 구현 논의</p>
5/8	<p>운영체제 4주차 진행상황 점검</p> <p><b>허성현:</b> 시스템 콜 추가 구현 계획</p> <p><b>윤재선:</b> slab allocator 구현 진행</p> <p><b>김도형:</b> xv6 스케줄링 방법 조사</p> <p><b>권정태:</b> xv6 시스템 콜 구현 방법 조사</p>
5/11	<p>운영체제 5주차 진행상황 점검</p> <p><b>허성현:</b> 시스템 콜 하나씩 구현 계획</p> <p><b>윤재선:</b> slab allocator 코드 디버깅</p> <p><b>김도형:</b> 스케줄링 구현 계획 및 성능 테스트 준비</p> <p><b>권정태:</b> xv6 설치 및 시스템 콜 구현 준비</p>
5/15	<p>구현 내용 정리 및 추후 계획</p>

	<p><b>허성현:</b> SJF 스케줄링 시스템 콜 구현</p> <p><b>윤재선:</b> slab allocator 기능 구현</p> <p><b>김도형:</b> SJF 스케줄러 구현</p> <p><b>권정태:</b> xv6 설치 및 시스템 콜 구현</p>
5/22	<p>구현 현황 점검 및 중간 보고서 작성</p> <p><b>허성현:</b> ps 함수 및 getnice 함수 구현</p> <p><b>윤재선:</b> slab allocator 테스트 및 디버깅</p> <p><b>김도형:</b> xv6 스케줄링 시스템 콜 구현</p> <p><b>권정태:</b> Function1 및 Function2 성능 테스트</p>
5/29	<p>slab_test.c 구현 및 최적화</p> <p><b>허성현:</b> kmalloc 및 kfree 함수 최적화</p> <p><b>윤재선:</b> slab_test.c 테스트 함수 작성</p> <p><b>김도형:</b> slab memory dump 함수 구현</p> <p><b>권정태:</b> 성능 테스트 및 결과 분석</p>
6/5	<p>slab_test.c 검토 및 디버깅</p> <p><b>허성현:</b> 메모리 할당 및 해제 과정 최적화</p> <p><b>윤재선:</b> 추가 테스트 케이스 작성</p> <p><b>김도형:</b> 문서화 작업 진행</p> <p><b>권정태:</b> 코드 리뷰 및 피드백 반영</p>

## ▼ 7. 결론

본 프로젝트는 xv6 운영체제의 메모리 관리 효율성을 크게 향상시키기 위해 다양한 크기의 메모리 할당기를 구현하고, 슬랩 할당 방식을 도입하는 데 중점을 두었습니다. 기존의 4KB 고정 크기 페이징 방식은 작은 크기의 메모리를 할당할 때 상당한 메모리 낭비를 초래했기 때문에, 이를 개선하여 메모리 사용의 효율성을 극대화하고자 했습니다.

우선, xv6 운영체제의 기본 동작과 메모리 관리 메커니즘을 철저히 분석했습니다. 이를 바탕으로, 8, 16, 32, 64, 128, 256, 512, 1024, 2048 바이트 크기의 메모리를 효율적으로 관리할 수 있는 슬랩 할당 방식을 설계하고 구현했습니다. 각 슬랩은 비트맵을 활용하여 객체의 할당 상태를 추적하고 관리함으로써, 객체 할당 및 해제 과정을 더욱 효율적으로 처리할 수 있었습니다.

### 프로젝트 성과

- 메모리 사용 효율성 증대:** 다양한 크기의 메모리 블록을 효율적으로 할당하고 내부 단편화를 최소화함으로써 메모리 자원의 낭비를 줄였습니다. 이를 통해 메모리 사용의 효율성을 크게 향상시켰습니다.
- 다양한 환경에서 프로그램이 실행될 수 있게 설계:** linux 환경에서 뿐만 아니라 docker 를 활용하여 docker와 visual studio가 설치 되어 있다면 어느 환경에서든지 실행될 수 있도록 하는 방법을 고안했습니다. 이를 통해 user는 복잡한 가이드를 따를 필요 없이 쉽게 프로그램을 실행 할 수 있다는 장점이 있습니다.
- 성능 최적화 및 신뢰성 확보:** `kmalloc` 및 `kfree` 함수를 구현하고 코드 level에서 최적화하였습니다. 다양한 테스트 케이스를 통해 구현된 메모리 할당기의 안정성과 신뢰성을 검증했습니다. 각 테스트 케이스는 메모리 할당, 해제, 재할당 등의 다양한 시나리오를 포함하여, 시스템이 다양한 상황에서도 안정적으로 동작함을 확인했습니다.

### 향후 연구 방향

이번 프로젝트를 통해 xv6 운영체제의 메모리 관리 방식이 크게 개선해 보았습니다. 다양한 크기의 메모리 할당기를 구현하여 메모리 낭비를 줄이고, 성능을 최적화함으로써 시스템의 전체적인 효율성을 높일 수 있었습니다. 프로젝트를 발전 시킨다면 user memory에서 slab을 할 당 할 때 효율적으로 할당 할 수 있게 스케줄링 방식 까지 바뀌보는 방향으로 진행할 예정입니다. 객체 크기가 작은 순서가 메모리 할당을 하는데 시간이 적게 걸리기 때문에 userlevel에서 큰 객체를 할당요청하고 나중에 작은 객체를 할당 하더라도 커널에서는 작은 메모리를 먼저 할당 되게끔 설계를 할 것입니다.